Revisiting the Combining Synchronization Technique

Panagiota Fatourou
Department of Computer Science
University of Crete & FORTH ICS
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
Department of Computer Science
University of Ioannina
nkallima@cs.uoi.gr

Abstract

Fine-grain thread synchronization has been proved, in several cases, to be outperformed by efficient implementations of the combining technique where a single thread, called the *combiner*, holding a coarse-grain lock, serves, in addition to its own synchronization request, active requests announced by other threads while they are waiting by performing some form of spinning. Efficient implementations of this technique significantly reduce the cost of synchronization, so in many cases they exhibit much better performance than the most efficient finely synchronized algorithms.

In this paper, we revisit the combining technique with the goal to discover where its real performance power resides and whether or how ensuring some desired properties (e.g., fairness in serving requests) would impact performance. We do so by presenting two new implementations of this technique; the first (CC-Synch) addresses systems that support coherent caches, whereas the second (DSM-Synch) works better in cacheless NUMA machines. In comparison to previous such implementations, the new implementations (1) provide bounds on the number of remote memory references (RMRs) that they perform, (2) support a stronger notion of fairness, and (3) use simpler and less basic primitives than previous approaches. In all our experiments, the new implementations outperform by far all previous state-of-the-art combining-based and fine-grain synchronization algorithms. Our experimental analysis sheds light to the questions we aimed to answer.

Several modern multi-core systems organize the cores into clusters and provide fast communication within the same cluster and much slower communication across clusters. We present an hierarchical version of CC-Synch, called H-Synch which exploits the hierarchical communication nature of such systems to achieve better performance. Experiments show that H-Synch significantly outperforms previous state-of-the-art hierarchical approaches.

We provide new implementations of common shared data structures (like stacks and queues) based on CC-Synch, DSM-Synch and H-Synch. Our experiments show that these implementations outperform by far all previous (fine-grain or combined-based) implementations of shared stacks and queues.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming, Parallel programming

General Terms Algorithms, Experimentation, Theory, Performance

Keywords Synchronization techniques, combining, hierarchical algorithms, blocking algorithms, concurrent data structures

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

1. Introduction

The last decade, the computer industry has made a significant turn towards developing multicore systems which nowadays, are used in any computing device. In such machines, increased performance can be achieved by exploiting parallelism; thus, harnessing the difficulty of concurrent programming is currently very important.

Several applications that are to be parallelized contain parts whose parallelization requires significant synchronization and coordination. Amdhal's law [1] implies that failing in parallelizing these parts may result in a significant limitation on the speed-up that could be achieved. However, these parts usually contain accesses to shared data and thus, parallelizing them requires the design of low-overhead synchronization mechanisms; without efficient such mechanisms the synchronization cost may overshadow any performance gain that could result from the parallelization of these parts.

Synchronization requests (e.g., accesses to the same shared data) must be executed in mutual exclusion; so, their execution often becomes a hot spot in a concurrent environment. Due to mutual exclusion, the best time that can be achieved to execute a number of such requests by any number of threads is no less than the time required by a single thread to execute them sequentially sidestepping the synchronization protocol. Therefore, ideally, the elapsed time to execute the same number of synchronization requests should be the same regardless of the number of threads that issue them. Apparently, this ideal behaviour is highly scalable and therefore very desirable. In practice, however, the contention effects may have a drastic impact in performance.

The *combining* synchronization technique [7, 12, 23] has recently gained ground since it has been experimentally proved that efficient implementations of this technique outperform all other synchronization mechanisms in several cases. In this technique, a list is employed to store the synchronization requests of active threads. After announcing its request by placing a node in the list, a thread tries to acquire a global lock. The thread that manages to acquire the lock, called the *combiner*, serves, in addition to its own synchronization request, active requests announced by other threads. In the meantime, each active thread that does not hold the lock, busy waits until either its request has been fulfilled by the combiner thread or the global lock has been released.

The combining technique has been employed for the design of common concurrent data structures like queues and stacks. Inter-thread communication is heavily based on accessing such concurrent data structures and therefore their efficient implementation is of major importance for achieving good performance and scalability. The combining-based implementations of such data structures

have experimentally been proved [7, 12] to outperform all previous state-of-the-art implementations of such structures which mainly employ some kind of fine-grain thread synchronization and take into consideration the data structure's access pattern.

In this paper, we revisit the combining technique aiming at discovering where its real performance power resides, understanding the performance implications of using different primitives when implementing it, and investigating whether and how ensuring some desired properties (e.g., fairness in serving requests) would impact performance. We do so by presenting two new implementations of this technique. The first, called CC-Synch, is suitable for *cache coherent* (CC) shared memory systems where accesses to shared objects are performed via cached copies of them; an access to a shared object is a *remote memory reference* (RMR) if the cached copy of this object is invalid, so the access causes a cache miss[1]. The vast majority of modern parallel architectures follow the CC shared memory model. The second implementation, called DSM-Synch, is better suited for the *distributed shared memory* (DSM) model, where a part of the shared memory is associated with each processor; so, each shared object is allocated (and resides) in the part of the shared memory that is associated to a specific processor. Processors do not have access to local caches, so a thread p performs a *remote memory reference* (RMR) if it accesses a shared object residing in the shared memory part of some processor other than that where p is being executed. Since an RMR is significantly more costly than a local memory reference [20], it is highly desirable to design algorithms that perform as few RMRs as possible; CC-Synch and DSM-Synch perform a bounded number of RMRs.

CC-Synch and DSM-Synch use a single FIFO queue to do both (1) implement the lock and (2) store the active synchronization requests. Therefore, the synchronization needed for implementing the list of active requests comes for free. Specifically, each newly activated thread adds a node to the tail of the queue to announce its request and participate to the implementation of the lock. Thus, each active thread is assigned one of the nodes of the queue. The active thread q that owns the first node of the queue becomes the combiner and undertakes the responsibility of applying some (or all) of the requests listed in the queue. Each active thread whose record is not first in the queue performs local spinning.

The experimental analysis (Section 3) reveals that the use of a highly-efficient queue-like lock which, in addition to its low synchronization overhead, provides the implementation of the list of announced requests for free, significantly reduces the synchronization required to implement the combining technique. Moreover, the new implementations are simpler to program than previous combining-based synchronization approaches. These result in a performance benefit in comparison to all previous combining-based synchronization approaches and in many cases to all fine-grain methods. Additionally, the new implementations exhibit several nice properties, not ensured by previous combining implementations [12, 23]. First, they provide stronger fairness guarantees in serving the requests. Second, they provide bounds on the number of remote memory references that are executed. Specifically, in CC-Synch, the combiner thread performs $O(h + t)$ RMRs, where h is an upper bound on the number of synchronization requests that the combiner may serve, and t is the size of the shared data that should be accessed in order to execute these h requests; we remark that h is a parameter that can be determined by the user and it can be chosen to be constant. The combiner in DSM-Synch performs $O(dh)$ RMRs, where d is the average number of RMRs required to serve a single request. In both algorithms, all threads, other than the combiner, perform local spinning and cause only a constant number of RMRs. Thus, the amortized number of performed RMRs is $O(d)$. Moreover, no thread may ever starve. Finally, the new implementations do not employ any form of backoff and they need minimal tuning to achieve the best performance.

CC-Synch uses a `Swap` object in addition to read-write (r/w) registers; a `Swap` object O supports in addition to `read`, the operation `Swap`(O, v) which (atomically) writes in O the value v and returns the previous value of O. DSM-Synch uses an object that supports `CAS` and `Swap` in addition to r/w registers; a `CAS`(O, u, v) (atomically) checks if the current value of O is u and if this is so, it changes the value of O to v and returns `TRUE`, otherwise the value of O remains unchanged and `FALSE` is returned. CC-Synch and DSM-Synch use just one primitive stronger than r/w registers and in CC-Synch this is a `Swap` object which is weaker than `CAS`. In CC-Synch, each thread maintains a single record to insert in the list, and therefore the total space overhead of CC-Synch is $O(n)$, where n is the number of threads; this is no more than that of previous combining-based synchronization approaches. The total space overhead for DSM-Synch is also $O(n)$.

We experimentally compare CC-Synch and DSM-Synch with several state-of-the-art synchronization approaches, like P-Sim [7], flat-combining [12], CLH spin locks [5, 18], and a simple lock free technique. The experiments (Figures 1 – 10) show that CC-Synch outperforms all these approaches in most cases. DSM-Synch outperforms all algorithms other than CC-Synch. DSM-Synch has the advantage over CC-Synch that it is designed to be efficient even in machines that support the DSM model; so, it can be executed efficiently by general scope applications which should run on different, not-necessarily predetermined architectures.

The experimental analysis reveals that the number of cache misses incurred per request is smaller in the new implementations than in previous techniques and the same is true for the cycles invested in memory stalls. Based on our experiments, we conclude that the technique of repeatedly performing CAS until it succeeds, even if it comes together with an appropriately-tuned back-off scheme, causes more cache misses and more branch mispredictions than employing Swap or other non-comparison primitives. Experiments also show that the average number of requests served by a combiner in CC-Synch and DSM-Synch is larger than in other algorithms, so the synchronization overhead paid to serve an amount of requests in these implementations is closer to the ideal than in previous approaches. So, the achieved combining degree has a significant impact on the performance of combining implementations.

We used CC-Synch and DSM-Synch to implement shared stacks and queues (Section 4). The stack implementation (CC-Stack) based on CC-Synch, outperforms all state-of-the-art shared stack implementations like the wait-free[2] stack implementation, called SimStack, presented in [7], the linked stack implementation based on flat-combining [12] where elimination has also been applied [13], and the stack implementation based on CLH spin locks [5, 18]. The stack implementation (DSM-Stack) based on DSM-Synch, outperforms all implementations other than CC-Stack. We also use CC-Synch and DSM-Synch to get two highly efficient shared queue implementations, called CC-Queue and DSM-Queue. More specifically, these implementations are derived by simply replacing the ordinary locks in the two-locks queue implementation presented by Michael and Scott in [21] with two instances of either CC-Synch or DSM-Synch. These implementations were experimentally compared to the wait-free queue implementation presented in [7] (called SimQueue), the two-locks

[1] Once the cache miss is served and as long as the data item is not updated by threads that are being executed on other processors, future accesses to the data item by threads that are being executed on this processor are local.

[2] An implementation is *wait-free*, if each thread completes the execution of an operation in a finite number of steps independently of the speeds or failures of other threads; wait-freedom is the strongest progress guarantee.

implementation [21], and the queue implementation based on flat-combining presented in [12]. CC-Queue performs up to 2.5 times faster than the queue implementation of [12] and outperforms SimQueue by a factor of up to 1.5.

For modern multi-core systems that organize the cores into clusters and provide fast communication (via shared caches) to the threads running in the same cluster and much slower communication across clusters, we present an hierarchical version of CC-Synch, called H-Synch, which exploits the hierarchical communication nature of such systems to achieve better performance. Experiments show that in such systems, H-Synch significantly outperforms CC-Synch and DSM-Synch as well as the state-of-the-art flat-combining NUMA locks recently presented by Dice *et. al* in [6]. We used H-Synch to design highly efficient implementations of stacks and queues for such machines. These implementations outperform by far, in such machines, CC-Stack, DSM-Stack, CC-Queue, and DSM-Queue, respectively, as well as all other concurrent stack and queue implementations with which these implementations have been compared.

CC-Synch and DSM-Synch are linearizable [15]. *Linearizability* is a well-accepted consistency condition for implementations of shared objects. It states that in any execution α of the implementation, each operation *op* on the simulated shared object executed in α appears to take effect, instantaneously, at some point, called the *linearization point* of *op*, in its execution interval.

Many hardware manufactures have been influenced by the universality result [14], and they have equipped their machines with the strongest atomic primitives (like CAS and LL/SC). As shown in [7], machines that additionally support Fetch&Add instructions, can have important performance advantages, while it is additionally possible to ensure wait-freedom. Our experiments show that machines that support Swap objects have significant performance benefits as well. Fortunately, Swap instructions are already supported by a large variety of machine architectures (x86, sparc, etc.).

We note that CC-Synch, similarly to Sim [7] and flat-combining [12], cannot be trivially applied in an efficient way for designing data structures such as search trees, where m lookups can be executed in parallel performing just a logarithmic number of shared memory accesses each. In such cases, it is expected that CC-Synch will perform well, only if several instances of it are employed. However, it is an interesting open problem to find efficient ways to synchronize these instances. We remark that using the combining technique to implement even simpler data structures, like shared linked lists, is also not obvious if several instances of the combining implementation should be employed to achieve good speed-up.

1.1 Related Work

Software combining was first realized in combining trees [8, 29]. To reduce the synchronization overhead of combining trees, a lot of research work has focused on designing adaptive versions of them [10, 20] (e.g., for implementing barriers) or decentralized algorithms for dynamically changing tree size [25, 26]. For some of these techniques [10, 20], it is not clear how they can be used to design general concurrent data structures, others [25] satisfy weaker consistency conditions than linearizability, and for others, experiments [12] have shown that their synchronization overhead is still high.

Oyama, Taura, and Yonezawa [23] present an implementation of the combining technique which we will call OyamaAlg. In OyamaAlg the lock is implemented using a CAS object O and the list of announced requests is implemented as a stack, so requests are served in a LIFO order. The CAS object is also used to maintain a pointer to the topmost element of the stack. To discover whether it is the combiner, a thread performs just a single CAS on O; however, to append its node in the stack, it repeatedly performs CAS until one of them is successful. So, the thread may starve. Before leaving the system, the combiner serves all other requests that have arrived in the system during the course of its execution, so the combiner may also starve. Finally, OyamaAlg has significant performance overheads for the following reasons. First, threads need to succeed on a CAS in order to have their requests announced; this causes a lot of contention and leads to a significant performance degradation. Second, the number of RMRs performed by a thread can be unbounded since threads may starve.

Flat-combining is another implementation of the combining technique presented by Hendler *et. al* in [12]. There are two main differences between flat-combining and OyamaAlg. In flat-combining, the list of announced requests usually contains one record for each thread independently of whether the thread has a currently active request; this reduces the number of insertions in the list. However, it increases the work of the combiner which should now traverse a longer list than necessary. To avoid this extra overhead, the combiner cleans up the list periodically keeping in it only the nodes of those threads that have recently initiated a request. Second, the CAS object O is not used to manipulate the head of the list as in OyamaAlg. This results in less overhead since the combiner does not interfere with threads that are trying to insert their records in the list. The combiner may choose to return without serving all the active requests in the list (so it does not starve), but this comes with a performance penalty since it makes it necessary to have each active thread checking regularly whether the coarse-grain lock has been released by the combiner and if yes, trying to become a combiner itself. In flat-combining, requests that have been inserted in the list later than other requests, may be served first. Moreover, flat-combining experiences some performance overheads. The cost paid by the combiner is larger than necessary if it traverses a list containing requests of currently non-active threads. Additionally, threads may perform an unbounded number of RMRs when trying to insert their nodes in the list.

Fatourou and Kallimanis have presented in [7] efficient universal constructions that implement the combining technique in a wait-free manner; a *universal* construction can be used to simulate any concurrent object. The first universal construction presented in [7], called Sim, performs a constant number of memory accesses, but it is only of theoretical interest since it uses unrealistically large objects. A practical version of Sim, called P-Sim, is also presented in [7]. P-Sim exhibits good performance in practice; it outperforms flat-combining (and other state-of-the-art synchronization techniques) in many cases. Efficient concurrent stack and queue implementations, based on P-Sim, are also presented and experimentally analyzed in [7]; in addition to being wait-free, these implementations significantly outperformed all previous state-of-the-art implementations of shared stacks and queues.

P-Sim does not easily cope with shared objects whose state size is large since it needs to copy locally the part of the state that it should be changed and then write back any updates. In contrast, the algorithms presented in this paper efficiently handle any shared object since the unique thread that applies the updates can do so directly on the shared data structure. More significantly, our experiments show that CC-Synch and DSM-Synch exhibit performance advantages over P-Sim. It is worth pointing out however that P-Sim has the benefit over CC-Synch and DSM-Synch that it ensures wait-freedom and therefore it is highly fault-tolerant.

Dice, Marathe and Shavit [6] have recently presented an hierarchical spin-lock implementation, called flat-combining NUMA lock; this hierarchical lock is based on flat-combining and exploits the cache hierarchies in order to provide good performance. As it is shown in [6], this lock greatly outperforms the previous (hierarchical and non-hierarchical) spin locks presented in [5, 17, 18, 20, 24]. Experiments show that H-Synch exhibits significant performance

Algorithm 1 Pseudocode for CC-Synch algorithm.

```
struct Node {
   Request req;
   RetVal ret;
   boolean wait;
   boolean completed;
   Node *next;
};
shared Node *Tail;
// Tail initially points to a dummy node
// with value ⟨⊥, ⊥,FALSE, FALSE, null⟩

// The following variable is private to each thread p_i; it is a pointer to a
// struct of type Node with initial value ⟨⊥, ⊥,FALSE, FALSE, null⟩
private Node *node_i;

RetVal CC-Synch(Request req) {     // Pseudocode for thread p_i
   Node *nextNode, *curNode, *tmpNode, *tmpNodeNext;
   int counter = 0;
1. nextNode = node_i;              // p_i uses a (possibly recycled) node
2. nextNode→next = null;
3. nextNode→wait = TRUE;
4. nextNode→completed = FALSE;
5. curNode = Swap(Tail, nextNode); // curNode is assigned to p_i
6. curNode→req = req;              // p_i announces its request
7. curNode→next = nextNode;
8. node_i = curNode;               // reuse this node next time
9. while (curNode→wait == TRUE)    // p_i spins until it is unlocked
     nop;
10. if (curNode→completed==TRUE)// if p_i's req is already applied
11.    return curNode → ret;       // p_i returns its return value
12. tmpNode = curNode;             // p_i is the combiner
13. while (tmpNode → next ≠ null AND counter < h) {
14.    counter = counter + 1;
15.    tmpNodeNext=tmpNode→next;
16.    apply tmpNode→req to object's state
         and store the return value to tmpNode→ret;
17.    tmpNode→completed = TRUE;// tmpNode's req is applied
18.    tmpNode→wait = FALSE;       // unlock the spinning thread
19.    tmpNode = tmpNodeNext;      // and proceed to next node
   }
20. tmpNode→wait = FALSE;          // unlock next node's owner
21. return curNode→ret;
}
```

advantages over flat-combining NUMA locks [6]. We believe that this is due to the fact that H-Synch (1) is simpler, (2) employs combining to serve the thread requests in each cluster (whereas this is not the case in the hierarchical lock presented in [6]), and (3) is based on CC-Synch which performs better than flat-combining.

CC-Synch implements a combining-friendly version of CLH [5, 18] queue lock; in contrast to CLH where the maintained queue is implicit, the queue maintained by CC-Synch is explicit so that the combiner can traverse it. DSM-Synch implements a combining-friendly version of MCS [20] queue lock.

2. Implementations of the Combining Technique

In this section, we present CC-Synch (Algorithm 1), H-Synch, and DSM-Synch (Algorithm 2).

2.1 CC-Synch

CC-Synch (Algorithm 1) maintains a list which contains one node for each thread that has initiated an active request; the list also contains a dummy node which is always the last node of the list. Each thread first announces its request by recording it in the last node of the list (i.e., in the dummy node of the list) and by inserting a new node as the last node of the list (which will comprise the new dummy node). At each point in time, we say that a node of the list *is assigned* to a thread p_i, if p_i has written the request recorded in the node; in CC-Synch, p_i is assigned the previous node to the node that it inserts in the list.

The thread that is assigned the head node of the list plays the role of the combiner, so it is the only thread that is allowed to access the shared data. The combiner starts by serving its own request. Other threads that have announced requests perform spinning on the $wait$ field of their assigned node. The combiner does not give up the lock when it completes the execution of its request; it rather continues accessing the next nodes of the list, it serves the requests announced in them, and sets their $wait$ field to FALSE in order to stop the threads that have been assigned these nodes from spinning. It also changes their $completed$ field to TRUE to identify that their requests have been completed. The combiner returns when it serves either all requests in the list or a pre-specified number h of such requests. In the later case, the combiner identifies the thread which owns the next to the last node that the combiner helps, as the new combiner; this is done by changing the $wait$ field of this node to FALSE while leaving its $completed$ field equal to FALSE.

We now give a more detailed description of CC-Synch. Pointer $Tail$ is a Swap object which initially points to a dummy node. Whenever thread p_i wants to announce a request req, it executes Swap on $Tail$ (line 5) in order to read the pointer to the dummy node pointed to by $Tail$ and update $Tail$ to point to its node (i.e., to the node pointed to by p_i's local variable $node_i$). Once this has been performed, p_i has been assigned the node that was previously pointed to by $Tail$, so it announces its request by recording req in the req field of this node (line 6) and then it sets the $next$ field of this node to point to the new dummy node (line 7). Next, p_i starts spinning on the $wait$ field of its assigned node until this field becomes FALSE. When p_i reads FALSE in $wait$, either its request has been executed by a combiner or p_i's record is the first in the list and therefore it owns the lock. In the former case, p_i simply returns (line 11), whereas in the later, p_i becomes the combiner.

Notice that the list could grow forever while the combiner thread p traverses it since a thread may add a node at the end of the list more than once after its request has been served by p. In order to prevent p from traversing a continuously growing list, an upper bound h (line 13) on the number of requests that p may serve is employed; once p serves h requests, it identifies the thread that has been assigned the next node of the list as the new combiner, and returns. Our experiments show that the choice of h does not significantly impact the performance of the algorithm. Specifically, setting h to a value equal to cn, where $c > 0$ is a small constant, is a good choice in terms of performance.

One criticism of CC-Synch could be that when a thread p executes Swap, it splits the list in two parts. If p is swapped out before executing line 7, all other threads can continue to add requests to the end of the list, but no thread can go over the list to execute the requests. However, several operating systems allow a thread to give a hint to the scheduler that it will soon execute a critical section [9] so that the scheduler avoids swapping out this thread. For instance, Solaris provides schedctl_start to announce that a thread will soon access the critical section, and schedctl_stop for informing the scheduler that the thread has finished executing the critical section. By using this technique, swapping out the combining thread can usually be avoided. We remark that schedctl_* is just a macro which executes a few assembly instructions with negligible performance cost.

Linearizability. CC-Synch is linearizable. Let req_i be any request initiated by thread p_i in an execution of CC-Synch. Let nd_i be the node of the list that is assigned to p_i for req_i. Thread p_i completes the execution of CC-Synch for req_i either on line 11 or

on line 21. Assume first, that p_i returns on line 11. In this case, a combiner thread p_j has served req_i before the execution of line 11 by p_i. Therefore, p_j has executed line 17 for nd_i at some iteration $l > 1$ of its while loop (lines $13 - 19$). Request req_i is linearized just before the execution of this instance of line 17 by p_j. Assume now that p_i returns on line 21. Then, p_i serves its request on its own when it executes line 16 at the first iteration of its while loop (lines $13 - 19$). In this case, req_i is linearized just before the execution of line 17 of the first iteration of p_i's while loop. Obviously, in both cases the linearization point of req_i is within its execution interval. Consistency is guaranteed since all requests are served sequentially. Due to lack of space, the complete proof of correctness will be provided in the full version of the paper.

RMR Complexity and Space Overhead. By the pseudocode (Algorithm 1), it follows that each thread returns either on line 11 or on line 21. If p_i returns on line 11, it obviously executes a constant number of RMRs. Assume now that p_i returns on line 21. By the pseudocode (line 13), p_i executes at most h iterations of the while loop (lines $14-19$). Lines $15-19$ contribute just a constant number of RMRs, and line 14 is executed on a local variable. Thus, p_i executes $O(h+t)$ RMRs, where t is the size of the shared memory data that they should be accessed in order to serve these h requests; we have assumed that the cache size of p_i's processor is greater than t. Notice that the amortized time complexity is $O(d)$, where d is the average number of RMRs required to serve a single request. We remark that in most cases, d equals a small constant. The space overhead of CC-Synch is $O(n)$, since each thread allocates one struct of type $Node$.

Required memory barriers. When implementing CC-Synch, memory barriers may need to be inserted in the code to ensure its correct execution. In architectures that implement either the TSO (Total Store Order) or the PO (Process Order) consistency model, we need to insert just one store memory barrier. These memory consistency models are very common and they are used in many contemporary multiprocessors, among which those that we used for our experiments. The first model is implemented on SPARC machines of version v8 and newer [28], while the second is implemented on AMD64 [22] and on Intel64 [4] architectures. SPARC processors support weaker consistency memory models as well, but they are rarely used and the TSO model is the default option for Solaris [19]. Both of these consistency models do not reorder two read operations, and the same holds for two store operations [19]. However, a read can be reordered with an older store only in case that the read and the store instructions access different memory locations [19]. Thus, for the correct execution of lines $6-7$ and lines $17-18$, no store barrier is needed. Similarly, no load barrier is needed just before line 10. A store memory barrier is inserted just before the return instruction of line 21. In cases where a weaker memory model is considered, additional memory barriers may have to be inserted; this is not the case in the architectures we employed for our experiments.

2.2 H-Synch

We now discuss how we can modify CC-Synch to get H-Synch. We consider a system of m processors which are partitioned into C clusters; each cluster consists of m/C processors. In such a system, communication among the processors of the same cluster is performed much faster than among processors residing in different clusters. A characteristic example of such a system is Niagara 2 in which we have executed some of the experiments in Section 3.

In H-Synch, the threads use C instances of CC-Synch one per cluster. Each instance of CC-Synch is used, as described in Section 2.1, to identify at each point in time, the combiner thread of each cluster and the list of announced requests of the threads that are executed at processors of the cluster. In addition to the C instances of CC-Synch, a queue lock L [5, 18] is used; L is accessed only by the combiner threads of the clusters. The CLH lock [5, 18] is a good choice for implementing L in systems where the intra-cluster communication is achieved with a cache coherent (CC) protocol, whereas the MCS lock [20] is expected to be a better choice in other systems.

Whenever a thread q has a newly activated request, it calls H-Synch. If q does not become the combiner of its cluster, it waits until its request has been served by a combiner of the cluster. Otherwise, before q starts serving requests, it executes an `acquire` operation on L in order to ensure that it is the only combiner (among those of the different clusters) that has access to the shared data. A combiner q serves only requests initiated by threads that are running on its local cluster. By doing so, intra-cluster communication is kept low. After finishing its work as a combiner, q releases L, so that a combiner of some other cluster can acquire L and have access to the shared data. In cases that the communication between clusters is performed through a more complex interconnection network (for instance, one that has an hierarchical structure), H-Synch can be easily modified to exploit the characteristics of the communication hierarchy by using more levels of queue locks (one queue lock per communication level). H-Synch ensures that requests initiated by threads of the same cluster are served in FIFO order and that the combiners acquire the global lock in FIFO order but, apparently, it does not globally ensure the FIFO property.

2.3 DSM-Synch

CC-Synch performs an unbounded number of RMRs in the DSM model. Here, we present DSM-Synch (Algorithm 2) which performs $O(hd)$ RMRs in this model. It maintains a list of announced requests which is updated in a manner similar to that of CC-Synch (and which also implements the lock). In contrast to CC-Synch, the list does not contain any dummy node and it is initially empty. Each thread p maintains two list nodes; p announces each request it wants to perform in one of these nodes, it inserts this node at the end of the list using `Swap`, and if its record is not the first in the list, it performs spinning on the $wait$ field of its node. If p's node is the first in the list, p becomes the combiner and serves up to h requests from those recorded in the list in FIFO order. A thread that wants to append a node updates the $next$ field of the previous node to point to the inserted node. The combiner serves requests recorded in list nodes up to the second last element of the list (see condition of the if statement of line 19). It does so to avoid accessing a node whose next field has not yet been updated although this node is not the last node in the list any more. Thus, a combiner will execute lines $22 - 25$ only if its node is the only node in the list. This ensures that a combiner performs a bounded number of RMRs.

We explain now why it is not enough to have each thread p_i using just one node. Let's assume that p_i has a single node that it reuses each time it initiates a new request. Let q be a combiner that serves p_i's request. Assume that there is a thread p_j whose assigned node is the next node of p_i's node in the list. Assume also that p_j's node is the last node in the list. After serving p_i's request, q sees that less than two nodes are left in the list and stops the execution of its while loop (lines $19-20$). Suppose that q stalls before executing line 26 of the pseudocode (pointer $tmpNode$ points to the node assigned to p_i). Assume now that p_i wants to immediately apply a new operation. Thus, p_i initializes its node again and inserts it at the end of the list by executing a `Swap` instruction on $Tail$. Then, q continues by executing line 26 of the pseudocode and makes an invalid memory reference.

Correctness & Memory Barriers. Using similar arguments as in CC-Synch, we can prove that DSM-Synch is linearizable. Specifically, consider a request req_i initiated by thread p_i, and assume that req_i is recorded in list node nd_i. As in CC-Synch, if

Algorithm 2 Pseudocode for DSM-Synch algorithm.

```
struct Node {
   Request req;
   RetVal ret;
   boolean wait;
   boolean completed;
   Node *next;
};

shared Node *Tail = null;

// the following variables are private to each thread p_i
private Node MyNodes_i[0..1] = {⟨⊥, ⊥, FALSE, FALSE, null⟩};
private int toggle_i = 0;

RetVal DSM-Synch(Request req) {     // pseudocode for thread p_i
    Node *tmpNode, *myNode, *myPredNode;
    int counter = 0;

1.  toggle_i = 1 - toggle_i;                // p_i toggles its toggle variable
2.  myNode=&MyNodes_i[toggle_i]; // p_i chooses to use one of its nodes
3.  myNode→wait = TRUE;
4.  myNode→completed = FALSE;
5.  myNode→next = null;
6.  myNode→req = req;                       // p_i announces its request
7.  myPredNode = swap(Tail, myNode); // p_i inserts myNode in the list
8.  if (myPredNode ≠ null) {                // if a node already exists in the list
9.     myPredNode→next = myNode;    // fix next of previous node
10.    while(myNode→wait == TRUE)  // p_i spins until it is unlocked
           nop;
11.    if(myNode→completed == TRUE)  // if p_i's req is already applied
12.       return myNode→ret;               p_i returns its return value
   }
13. tmpNode = myNode;
14. while(TRUE) {                           // p_i is the combiner
15.    counter++;
16.    apply tmpNode→req to object's state
         and store the return value to tmpNode→ret;
17.    tmpNode→completed = TRUE;   // tmpNode's req is applied
18.    tmpNode→wait = FALSE;          // unlock the spinning thread
19.    if (tmpNode→next == null or
           tmpNode→next→next == null or counter ≥ h)
20.       break;      // p_i helped h threads or fewer than 2 nodes are in list
21.    tmpNode = tmpNode→next;      // proceed to the next node
   }
22. if (tmpNode→next == null) {      // p_i's req is the single record in list
23.    if(CAS(Tail, tmpNode, null)==TRUE)   // try to set Tail to null
24.       return myNode→ret;
25.    while (tmpNode→next == null)// some thread is appending a node
           nop;                             // wait until it finishes its operation
   }
26. tmpNode→next→wait = FALSE;   // unlock next node's owner
27. tmpNode→next = null;
28. return myNode→ret;
}
```

p_i returns on line 12, req_i is linearized just before the execution of line 18 for nd_i by the combiner; if p_i returns on line 28, req_i is linearized just before the execution of line 18 at the first iteration of the while loop executed by p_i. As in CC-Synch, when implementing DSM-Synch, a store memory barrier is inserted in the code just before line 28 where the algorithm returns.

RMR Complexity and Space Overhead. DSM-Synch performs $O(hd)$ RMRs when it executes the while loop of line 14. The only extra piece of code that may cause DSM-Synch to perform more RMRs is that of lines $22-25$. However, as explained above a thread p executes these lines only when the node it inserts is the single node in the list. So, if these lines are executed, p's local variable called $tmpNode$ is equal to $myNode$, and therefore spinning on $tmpNode \rightarrow next$ on line 25 is local. The space overhead of DSM-Synch is $O(n)$.

3. Performance evaluation

We evaluated CC-Synch and DSM-Synch in two different multiprocessor machine architectures. The first is a 32-core machine consisting of four AMD Opteron 6134 processors (Magny Cours). Each processor consists of 2 dies and each die contains 4 processing cores. Communication among the cores of the same die is achieved with a fast L3 cache. Dies communicate with HyperTransport links which create a complex topology that resembles a hypercube [3]. The second machine is a 128-way Sun consisting of 2 UltraSPARC-T2 processors (Niagara 2). Each processor consists of 8 processing cores, each of which is able to handle 8 threads. Communication among the cores of the same processor is achieved with a fast L2 cache. All experiments on the Magny Cours machine were performed using the gcc $4.3.4$ compiler, while experiments on the Niagara 2 machine were performed using gcc $4.5.1$. In order to avoid bottlenecks in memory allocation, the Hoard memory allocator [2] was used. The operating system running on the Magny Cours machine was Linux with kernel $2.6.18$, while the operating system running on the Niagara 2 machine was Solaris 10.

Thread binding is employed for the following reason. Assume that the number of threads is smaller than the number of cores and suppose that two threads are running. The scheduler may decide to run them either on different processors (chips) or within the same chip. In the first case the communication cost is an order of magnitude more than in the second. Thus, if thread binding is not used, a significant uncertainty factor is introduced which may lead to an unreliable experiment. We observed that this was a usual phenomenon. So, on the Magny Cours machine, the i-th thread was bound to the i-th core of the machine; we first exploited multi-core, then multi-chip and then multi-socket configuration. On the Niagara 2 machine, we follow a slightly different scheduling similar to that used in [6] in order to better explore the performance properties of hierarchical algorithms. More specifically, we split threads into two groups, one for each socket.

In order to evaluate CC-Synch and DSM-Synch, we compare their performance with that of state-of-the-art synchronization techniques. Specifically, they are compared with P-Sim (the wait-free universal construction presented in [7]), flat-combining [11, 12], the CLH spin-lock [5, 18][3], OyamaAlg [23], and a simple lock-free implementation. The lock-free implementation was implemented using a `CAS` object. Specifically, whenever a thread wants to apply a `Fetch&Multiply`, it repeatedly executes `CAS` until it succeeds; a backoff scheme is employed to increase the scalability of this implementation. Since the Niagara 2 machine does not support `Fetch&Add` which is employed by P-Sim and is necessary, as shown in [7], for its good performance, no experiment was performed for P-Sim on the Niagara 2 machine. We also evaluated a variation of CC-Synch (called CAS-Synch), in which `Swap` is simulated with a `CAS` object in a lock-free manner. This allows us to explore the performance advantages of `Swap` over `CAS`.

On the Niagara 2 machine, H-Synch and the hierarchical NUMA lock (called FC-MCS below) recently presented in [6] were also evaluated. On the Magny Cours machine, experiments show no performance benefit when using any of the hierarchical techniques. This is rational to the fact that the machine consists of many but very small clusters of cores. Thus, we have not included any performance measurements for the hierarchical algorithms on the Magny Cours machine. All algorithms were carefully optimized and for those that use backoff schemes, we performed a large number of experiments in order to choose the best backoff parameters. We used the flat-combining implementation

[3] As expected for cache-coherent NUMA architectures, we experimentally saw that MCS [20] spin locks have slightly worse performance than CLH locks in both machines, so we present experimental results for CLH locks.

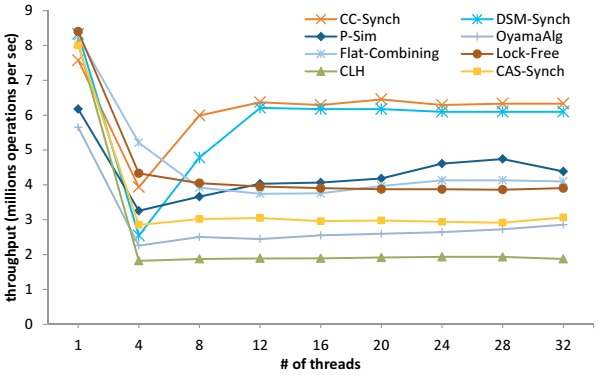

Figure 1: Average throughput of each implementation on the Magny Cours machine while simulating a `Fetch&Multiply` object.

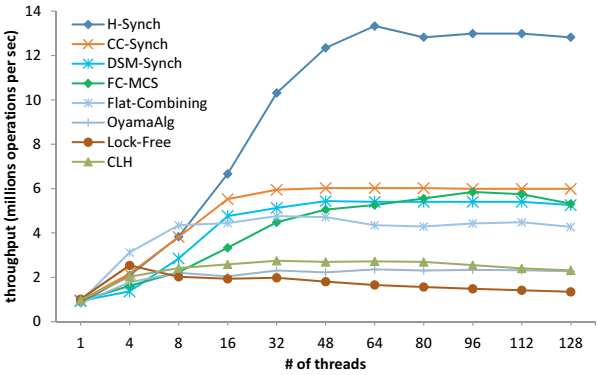

Figure 2: Average throughput of each implementation on the Niagara 2 machine while simulating a `Fetch&Multiply` object.

that was provided by its inventors [11, 12] and we choose its parameters very carefully in order to achieve the best performance. We further optimized the code of flat-combining to run faster than its original version [11, 12] on the Magny Cours machine. We used the latest version of P-Sim code (version 0.8) [7, 16]. The source code of our implementations is provided at `http://code.google.com/p/sim-universal-construction/`.

The first experiment we performed is a synthetic benchmark (Figures 1, 2), where a simple `Fetch&Multiply` object is simulated. We measure the average throughput (`Fetch&Multiply` per second) that each synchronization technique achieves when it executes 10^7 `Fetch&Multiply` operations (i.e., always the same amount of work), for different values of n; each thread executes $10^7/n$ `Fetch&Multiply`. Specifically, the horizontal axis of Figures 1, 2 represents the number of threads n, while the vertical axis displays the throughput (in millions of operations per second) that each synchronization technique has performed. For each value of n, the experiment has been performed 10 times and averages have been calculated. A random number of dummy loop iterations (up to 64) have been inserted between the execution of two `Fetch&Multiply` by the same thread; specifically, in each iteration a volatile counter is increased. In this way, we simulate a random work load large enough to avoid unrealistically low cache miss ratios and long runs (but not too big to reduce contention). Figure 6 studies the performance behavior of our algorithms for different values of the random work.

In the experiments performed on the Magny Cours machine (Figure 1), CC-Synch outperforms all other synchronization techniques. Specifically, CC-Synch achieves up to 1.54 higher throughput than flat-combining and outperforms P-Sim by a factor of up to 1.52. The lock free implementation of `Fetch&Multiply` is slightly slower than P-Sim and flat-combining. Also, CC-Synch is up to 2.7 times faster than OyamaAlg [23]. DSM-Synch performs also very well; its performance is close to that of CC-Synch, despite the fact that it is designed for machines following the DSM model. Figure 1 also shows that simulating `Swap` using `CAS` (in a lock-free way) induces a serious performance penalty; specifically, CAS-Synch is two times slower than CC-Synch.

Similarly to the experiments performed on the Magny Cours machine, CC-Synch outperforms all algorithms other than H-Synch on the Niagara 2 machine (Figure 2). More specifically, CC-Synch outperforms flat-combining by a factor of up to 1.4. It is noticeable that even CC-Synch itself (not its hierarchical version) outperforms FC-MCS [6] by a factor of up to 1.65, despite the fact that FC-MCS exploits the hierarchical characteristics of communication in the machine. The relatively small performance gap between flat-combing and FC-MCS may seem surprising at first; however, this result is rational to the fact that FC-MCS causes more cache misses when accessing the shared data. Specifically, in FC-MCS, combining is not used in applying the requests, so each request may be applied by a different thread; thus, each time a thread accesses the shared data cache misses may occur. This is avoided when combining is employed in serving the requests, as done by the other studied combining-based synchronization techniques. DSM-Synch exposes almost the same performance to CC-Synch on the Niagara 2 machine. H-Synch which is the hierarchical version of CC-Synch outperforms FC-MCS by a factor of up to 2.65 and flat-combining itself by a factor of up to 3.0. CC-Synch is up to 2.55 times faster than OyamaAlg [23], while H-Synch is more than 6 times faster. The performance of CAS-Synch is not illustrated in Figure 2 since CAS-Synch results in very poor performance on the Niagara 2 machine.

As shown in Figure 1, on the Magny Cours machine, all algorithms perform faster in case $n = 1$ than for larger values of n. On the contrary, Figure 2 shows that, on the Niagara 2 machine, the performance of all algorithms is always better for larger values of n. This is due to the fact that the Magny Cours machine implements atomic instructions (`CAS`, `Swap`, etc.) in the private L1 cache which is very fast; in contrast, a Niagara 2 processor implements atomic instructions in the shared L2 cache which is slower (Niagara 2 processor is optimized for contented workloads, i.e. in case of $n > 1$).

In Figure 3, we study the performance of each implementation on the Niagara 2 machine for $n > 128$, i.e., when n is larger than the number of threads that the machine is able to handle simultaneously; thus, the machine is oversubscribed. We do not include any measurement from FC-MCS and CLH locks since in this experiment they do not achieve good performance. As illustrated in Figure 3, H-Synch, CC-Synch, and DSM-Synch achieve better performance than any other synchronization technique for any value of n.

Figures 4 - 5 aim at investigating the reasons for the good performance of CC-Synch and DSM-Synch. More specifically, from Figure 5, it follows that on the Magny Cours machine, P-Sim and flat-combining execute slightly more (up to 10% more) atomic instructions than CC-Synch and OyamaAlg [23]. The experiments showed that to achieve the best performance for the lock-free algorithm, the back-off should not be too high. By appropriately choosing the back-off to get the best performance, it turned out that the average number of CAS performed by each instance of the algorithm is two which is bigger than the average number of atomic instructions executed by each instance of CC-Synch and DSM-Synch. Thus, the lock-free algorithm has a performance disadvantage compared to these algorithms.

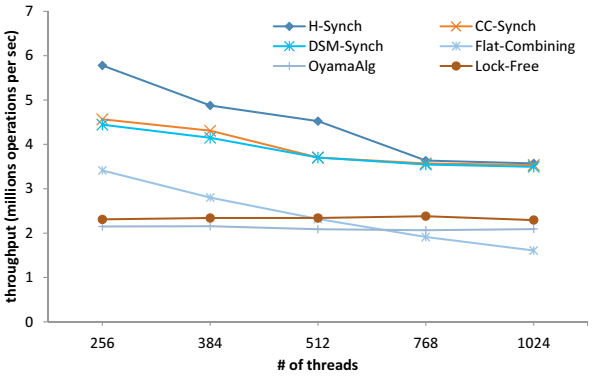

Figure 3: Average throughput of each implementation for $n > 128$ on the Niagara 2 (oversubscribing) machine while simulating a `Fetch&Multiply` object.

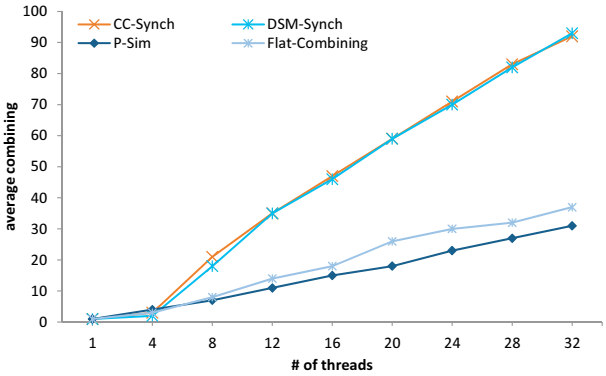

Figure 4: Average degree of combining of each implementation while simulating a `Fetch&Multiply` object.

Figure 4 displays the average degree of combining, i.e., the average number of requests that are executed by a combiner. It shows that CC-Synch and DSM-Synch achieve better degree of combining which is almost 3 times more than that of P-Sim and flat-combining[4]. Our efforts to increase the average degree of combining for flat-combining by carefully tuning its parameters (i.e., by increasing the combining rounds or by changing the polling level), revealed that when the combining degree was increased the average throughput was decreased. On the contrary, P-Sim operates in a way that it can help as much threads as the system's point contention (i.e., as the maximum number of threads that can be simultaneously active at any point in time which might be equal to n).

In the experiment of Figure 6, we studied the behavior of the competing algorithms for different amounts of random work. This experiment was executed on the Magny Cours machine; the number of threads was fixed to 32. Figure 6 shows that for a wide range of values for random work ($64 - 2048$), most algorithms have a small difference in the exhibited throughput. This shows that the communication cost is the dominant factor, whereas the time invested to execute the local random work does not play any significant role. An exception is the lock-free algorithm, which, in case the random work is equal to zero, has unrealistically high performance. This is due to the fact that, in this case, a thread could uninterruptedly execute thousands of `Fetch&Multiply` before some other thread

[4] In Figure 4, we have not included results for OyamaAlg since the variance of the combining degree in this algorithm was too high to get a realistic view. This is due to the fact that a combiner thread in this algorithm may be enforced to help an unbounded number of operations.

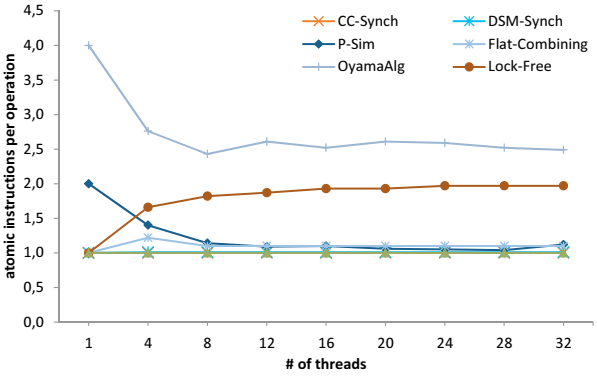

Figure 5: Average number of atomic instructions (`CAS`, `Swap`, `Fetch&Add`) that each implementation executes while simulating a `Fetch&Multiply` object.

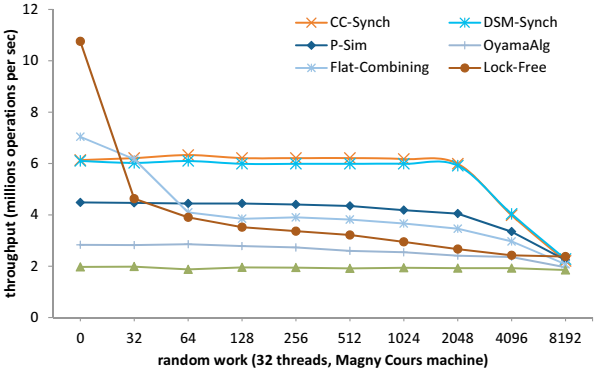

Figure 6: Average throughput of CC-Queue and DSM-Queue on the Magny Cours machine, with different values of random work, $n = 32$.

starts its execution. This phenomenon (called long runs) has been also discussed in [7, 21] as an unrealistic workload. Figure 6 shows that by slightly increasing the random work, the performance of the lock-free algorithm rapidly decreases. The same phenomenon, although in a smaller scale, was also noticed for flat-combining. Similarly to the lock-free implementation, flat-combining has high throughput for very small amounts of random work, although its performance vastly decreases when the random work is slightly increased. In cases that the number of iterations is 2048 or more, the time needed to execute this loop becomes the dominant performance factor, i.e., executing the loop becomes more expensive than executing the algorithm for applying a FetchAndMultiply instruction. Thus, the performance of all algorithms starts to decrease and the performance gap between them becomes insignificant. We remark that the scale of the horizontal axis of Figure 6 is logarithmic.

Algorithm	cache misses	cpu cycles spent in memory stalls
CC-Synch	4.1	2747
Sim	4.9	6328
flat-combining	5.8	6501

Table 1: Cache misses and memory stalls per operation, $n = 16$.

Table 1 shows some measurements from performance counters. We observed that the extra cache misses incurred by P-Sim and flat-combining was caused due to the number of failed `CAS` instructions; this number becomes worse if these algorithms are not properly tuned. Since failed `CAS` instructions cause cache misses and branch mispredictions, we conclude that a combining technique that avoids them has a serious performance advantage.

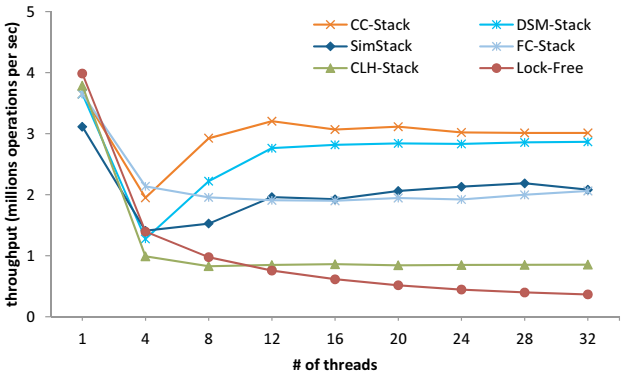

Figure 7: Average throughput of the CC-Stack and DSM-Stack on the Magny Cours machine.

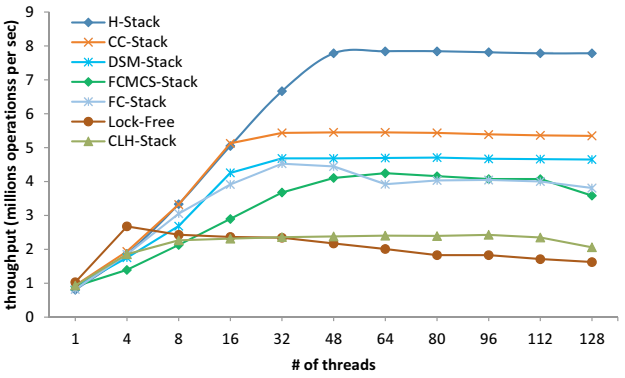

Figure 8: Average throughput of H-Stack, CC-Stack and DSM-Stack on the Niagara 2 machine.

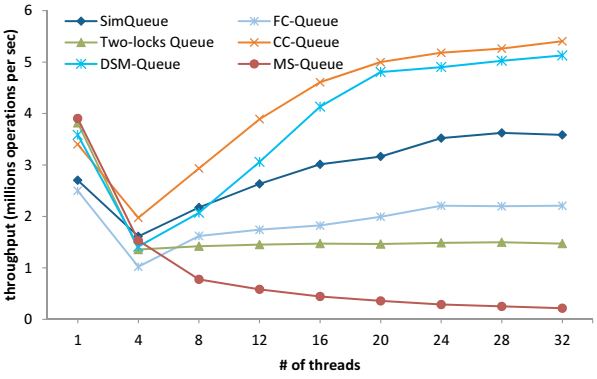

Figure 9: Average throughput of the CC-Queue and DSM-Queue on the Magny Cours machine.

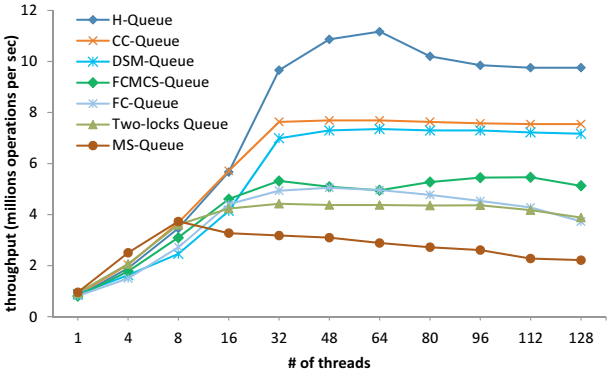

Figure 10: Average throughput of H-Queue, CC-Queue and DSM-Queue on the Niagara 2 machine.

4. Highly Efficient Data Structures

We further investigate the performance of CC-Synch, DSM-Synch, and H-Synch by implementing common concurrent data structures (i.e., shared stacks and queues). We compare the performance of these implementations with state-of-the-art shared stack and queue implementations. Specifically, the shared stack implementation based on CC-Synch, called CC-Stack, was evaluated against SimStack [7], the lock-free stack implementation presented by Treiber in [27], a stack implementation based on CLH spin locks [5, 18], and a linked stack implementation based on flat-combining [11, 12] (called FC-Stack). Both CC-Stack and FC-Stack eliminate pairs of push and pop whenever possible; the performance of elimination [13] has been studied in [7] and [12] where experiments show that elimination is outperformed by SimStack and FC-Stack. We also implemented a shared stack based on H-Synch and FC-MCS [6] and evaluated their performance on the Niagara 2 machine.

The experiment we perform is similar to that performed by Michael and Scott for queues in [21]. We measure the average throughput (operations per second) that each algorithm achieves (every thread executes $10^7/n$ pairs of ENQUEUE and DEQUEUE operations) for different values of n. Again, the experiments have been performed several times and averages have been taken; we have simulated a random workload by executing a random number of iterations (up to 64) of a dummy loop after each operation.

As it is shown in Figure 7, on the Magny Cours machine, CC-Stack performs up to 1.68 times faster than FC-Stack, and up to 1.59 times faster than SimStack. The stack implementation based on the CLH spin lock had much lower performance. The stack implementation based on DSM-Synch, called DSM-Stack, performs slightly worse than CC-Synch but it is much better than all other algorithms. On the Niagara 2 machine, the shared stack based on CC-Synch performs 1.4 times faster than FC-Stack (Figure 8). The stack implementation based on DSM-Synch performs worse than CC-Stack but it is again better than all other algorithms. It is noticeable that the stack implementations based on CC-Synch and DSM-Synch outperform by a factor of up to 1.49 the shared stack based on FC-MCS of [6]. The stack implementation based on H-Synch significantly outperforms all other implementations, being up to 2.0 times faster than FC-Stack and up to 2.1 times faster than the stack based on FC-MCS [6].

We also implement and experimentally analyze shared queues based on CC-Synch, DSM-Synch, and H-Synch (called CC-Queue, DSM-Queue, and H-Queue, respectively). Specifically, the two-locks queue implementation presented in [21] is enhanced by replacing the ordinary locks with instances of CC-Synch, DSM-Synch, and H-Synch, respectively. These implementations are compared (Figures 9-10) with SimQueue [7], the lock free queue implementation and the two-locks implementation presented in [21], and the queue implementation based on flat-combining [11, 12] (called FC-Queue). On the Niagara 2 machine, we additionally implemented and evaluated a two-locks queue variant using FC-MCS [6]. The queue experiment was similar to that for stacks.

As illustrated in Figure 9, on the Magny Cours machine, SimQueue exhibits better performance than any algorithm other than CC-Queue and DSM-Queue, as expected based on results in [7]. CC-Queue performs up to 2.53 times faster than FC-Queue (Figure 9) and 2.1 times faster than SimQueue. DSM-Queue per-

forms slightly worse than CC-Queue but better than all other algorithms. On the Niagara 2 machine (Figure 10), FC-Queue performs better than all algorithms other than CC-Queue, DSM-Queue, and H-Queue (recall that SimQueue has not been implemented in this machine since `Fetch&Add` instructions are not included in its instruction set). CC-Queue performs up to 1.8 times faster than FC-Queue and up to 1.55 times faster than the queue based on [6]. The queue implementation based on H-Synch greatly outperforms all other candidates by being up to 2.25 times faster than the queue implementation based on FC-MCS [6]. It is also noticeable that the performance gap between FC-Queue and the two-locks queue is smaller on the Niagara 2 machine. This is due to the fact that the CLH locks perform very well in this machine and the parallel use of two different locks (one for enqueues and one for dequeues) gives a performance boost in the two-locks algorithm. Again, DSM-Queue performs slightly worse than CC-Queue but better than all other algorithms except from H-Queue on the Niagara 2 machine.

Acknowledgments

We would like to thank Dimitris Nikolopoulos for arranging the provision of access to some of the multi-core machines of the CS Dept. of Virginia Tech where we ran our experiments. Many thanks also to Michael Scott for providing access to the Rochester's Niagara 2 machine. We would like to especially thank Victor Luchangco and Faith Ellen for several fruitful discussions and for their valuable feedback on the paper. Thanks also to Nir Shavit for several interesting discussions on a preliminary version of this paper and to the anonymous reviewers for their interesting comments on the submitted version of the paper.

Nikolaos Kallimanis is supported by a PhD scholarship from the Empirikion Foundation, Athens, Greece. The work of Panagiota Fatourou was supported by the European Commission under the 6th and 7th Framework Programs through the TransForm (FP7-MC-ITN-238639), Hi-PEAC2 (FP7-ICT-217068), and ENCORE (FP7-ICT-248647) projects.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, page 483485, 1967.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2000.

[3] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Blade Computing with the AMD Opteron Processor (Magny-Cours). *Hot chips 21*, August 2009.

[4] I. Corporation. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part1*, January 2011.

[5] T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, February 1993.

[6] D. Dice, V. J. Marathe, and N. Shavit. Flat-Combining NUMA Locks. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 65 – 74, 2011.

[7] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325 – 334, 2011.

[8] J. R. Goodman, M. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, April 1989.

[9] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. *SIGMETRICS Perform. Eval. Rev.*, 19:120–132, April 1991.

[10] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 54–63, 1989.

[11] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. The code for Flat-Combining. http://github.com/mit-carbon/flat-combining.

[12] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 355–364, 2010.

[13] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th ACM Symposium on Parallel Algorithms and Architectures*, pages 206–215, 2004.

[14] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, jan 1991.

[15] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, 1990.

[16] N. D. Kallimanis and P. Fatourou. The code for sim universal construction. http://code.google.com/p/sim-universal-construction/.

[17] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Euro-Par Conference*, pages 801–810, 2006.

[18] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 165–171, 1994.

[19] P. E. McKenney. *Memory Barriers: a Hardware View for Software Hackers*, June 2010.

[20] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[22] A. Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, June 2010.

[23] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99)*, pages 182 – 204, 1999.

[24] Z. Radovic and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th IEEE International Symposium on High-Performance Computer Architecture*, pages 241–252, 2003.

[25] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[26] N. Shavit and A. Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[27] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[28] D. L. Weaver and T. Germond. *The SPARC Architecture Manual, Version 9*, 1994.

[29] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Computers*, 36(4):388–395, 1987.